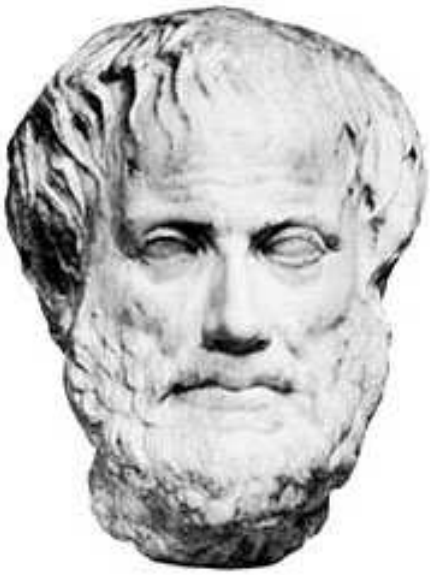
The meaning of "to be" in computer programming

- Aliases,
- Shared references,
- Pointer Equality, and
- Structural Equality

‘Is’, ‘To be or...’, ‘Is a’



Is, is a ...

"What the meaning of the word 'is' is..." - Bill Clinton

"To be or not to be..." - Hamlet (William Shakespeare)

If every A has a B and x *is an* A then x has a B - (Aristotlean silogism)

(If every city has a mayor, and Edinburgh is a city, then Edinburgh has a mayor)

- Being something
- Being a kind of something
- Being the same as something
- Being equal to something else
- Sameness vs equality

Being something

- Variables and values
- If we execute:

```
x = 5;
```

- then the value of `x` is 5.
- Strictly speaking `x` is not 5; `x` is a memory location.
- So while we would informally read `x==5` as "`x` is 5", the actual meanning is the value of `x` is 5.
- Hence, after executing

```
x = 5;
y = 5;
```

and both `x` and `y` have the same value, but they are not the same variable.

Variables and values

- For primitive data types (int, boolean, float, String, etc.)

```
x = y;
```

means copy the value of y in the memory location of x;

- So

```
int x, y;
x = 4;
y = x;
```

means that both `x` and `y` have value `4`, but they have a separate identity because each of them is a different memory location...

x	4
y	4

Variables and values

- So the value of `y` is the same as the value of `x`, but `y` is not the same as `x`.
- ... which implies that their values are independent:

```
int x, y;
x = 4;
y = x;
x++;
// x == 5 and y == 4
```

- Variables can be changed over time by assignment.
- If `x` and `y` are two variables of a primitive data type, we say that they are equal if their values are the same.
- We can test for whether the values of two variables are the same using the == operator.

Being of some kind

- The "is a" relationship between an object (or instance) and its class
- So if we have a class

```
class A {
  //...
}
```

- and in some client code we have

```
A x;
x = new A();
```

- Then `x` is an `A`.
- The variable `x` is of type `A`
- The value of `x` is an object of type `A`
- The object referred to by `x` is a kind of `A`.

Being the "same" as something

- Suppose we have

```
A x, y;
x = new A();
y = new A();
```

- Both variables `x` and `y` are `A`'s
- ... but the objects they refer to are different, individual, and independent `A`'s.

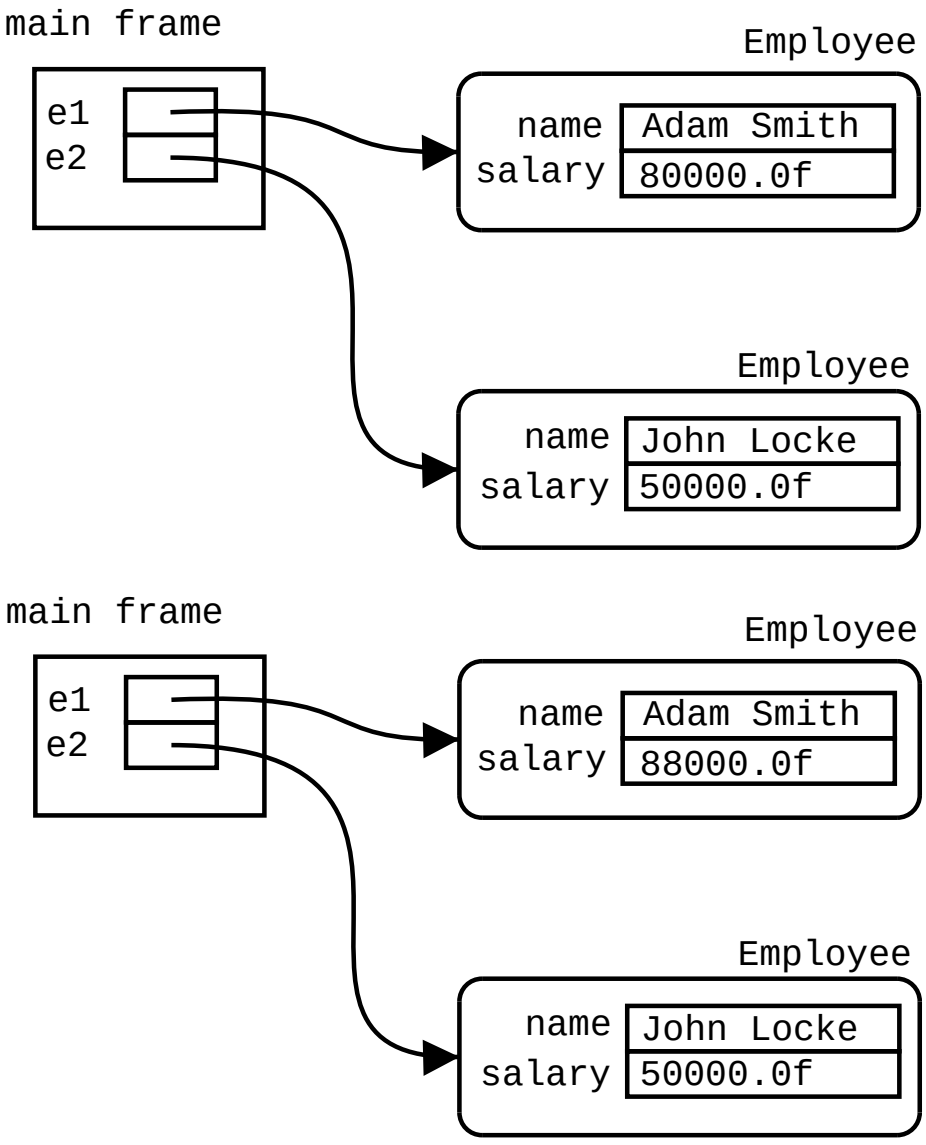
Example:

```
class Employee
{
    String name;
    float salary;
    Employee(String name, float salary)
    {
        this.name = name;
        this.salary = salary;
    }
    String name() { return name; }
    float salary() { return salary; }
    void raise_salary(float percentage)
    {
        salary = salary * (1 + percentage/100.0f);
    }
}
```

Example (contd.)

```
public class Test
{
  public static void main(String[] args)
  {
    Employee e1 = new Employee(“Adam Smith”, 80000
    Employee e2 = new Employee(“John Locke”, 50000
    e1.raise_salary(10f);
    System.out.println(e2.salary());
  }
}
```

Example (contd.)



Alias

- A variable is an alias of another variable if they both point to the same object.

```
A x, y;
x = new A();
y = x;
```

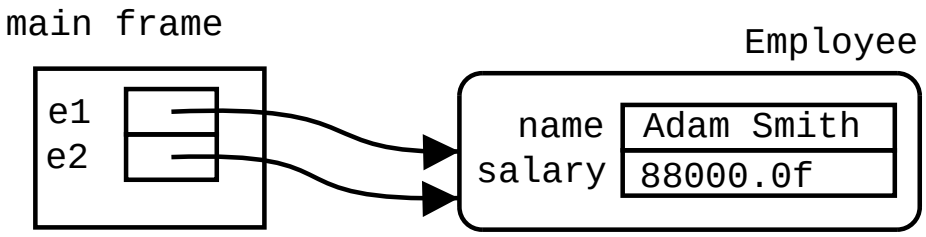
- In this case `x` and `y` are the "same".

- More precisely, the values of `x` and `y` are the same reference (pointer,) and therefore they refer to the same object.

Example (contd.)

```
public class Test
{
  public static void main(String[] args)
  {
    Employee e1 = new Employee(“Adam Smith”, 80000
    Employee e2 = e1;
    e1.raise_salary(10f);
    System.out.println(e2.salary());
  }
}
```

Example (contd.)



Aliases

- Compare Test with

```
int x1, x2;
x1 = 6;
x2 = x1;
x1 = x1 * 3;
```

- If two variables are aliases, whatever one does to either of them, affects the other, because they refer to the same object.

Shared references

```
public class BankAccount
{
  private float balance;
  public BankAccount(float b) { balance = b; }
  public void deposit(float amount)
  {
    balance = balance + amount;
  }
  public void withdraw(float amount)
  {
    if (balance >= amount)
      balance = balance - amount;
  }
  public float balance() { return balance; }
}
```

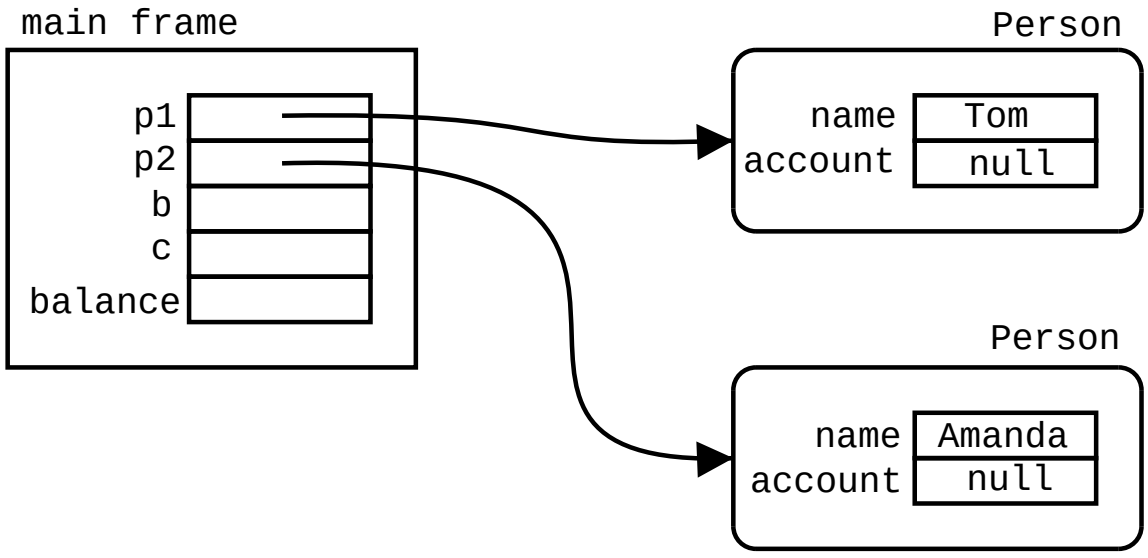
Shared references

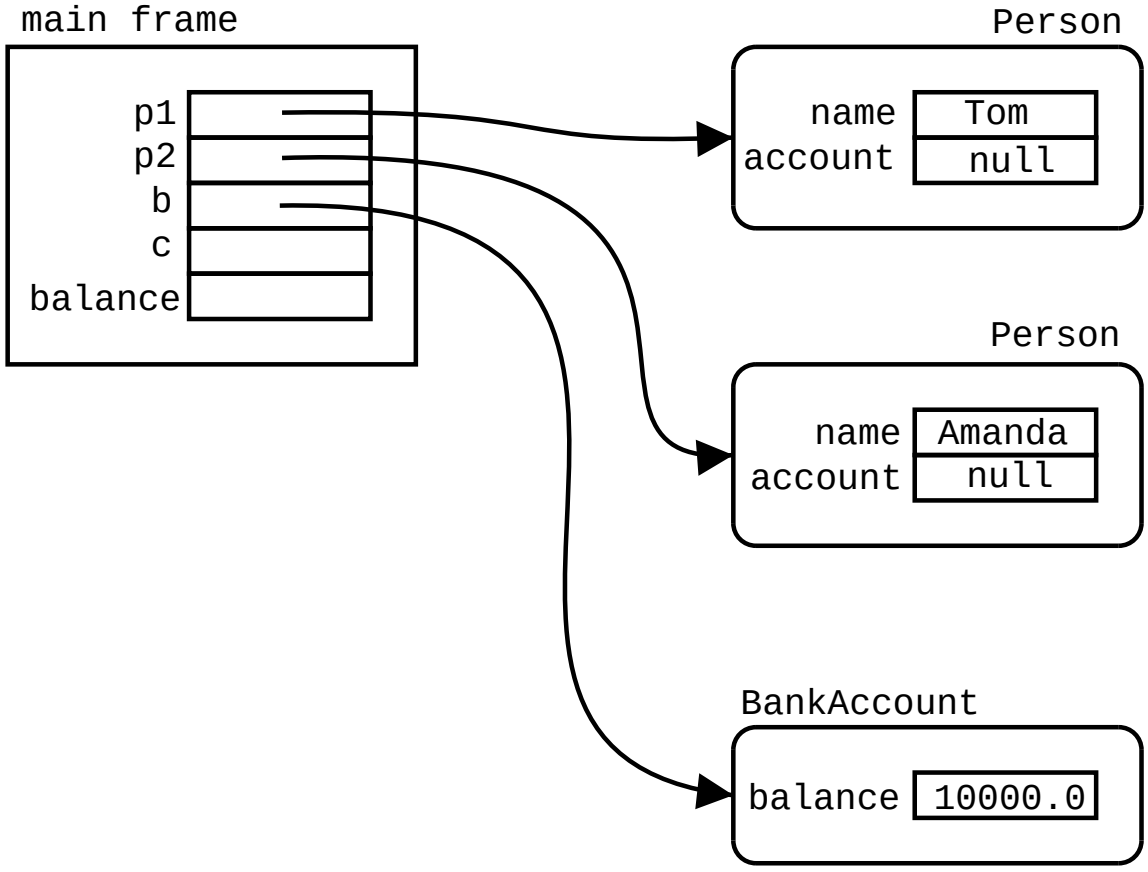
```
public class Person
{
  private String name;
  private BankAccount account;
  public Person(String name) { this.name = name; }
  public void set_account(BankAccount a)
  {
    account = a;
  }
  public String name() { return name; }
  public BankAccount account() { return account; }
}
```

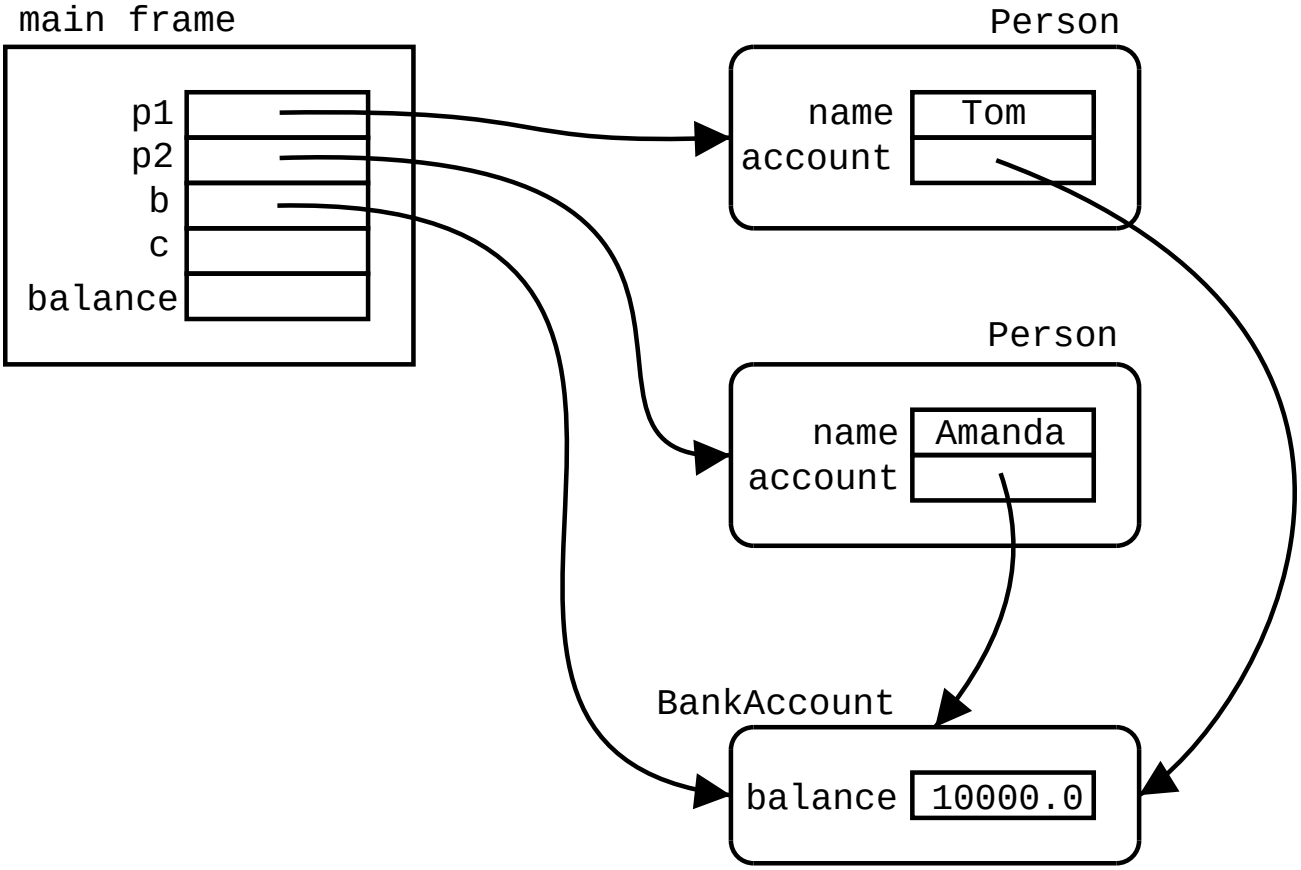
Shared references

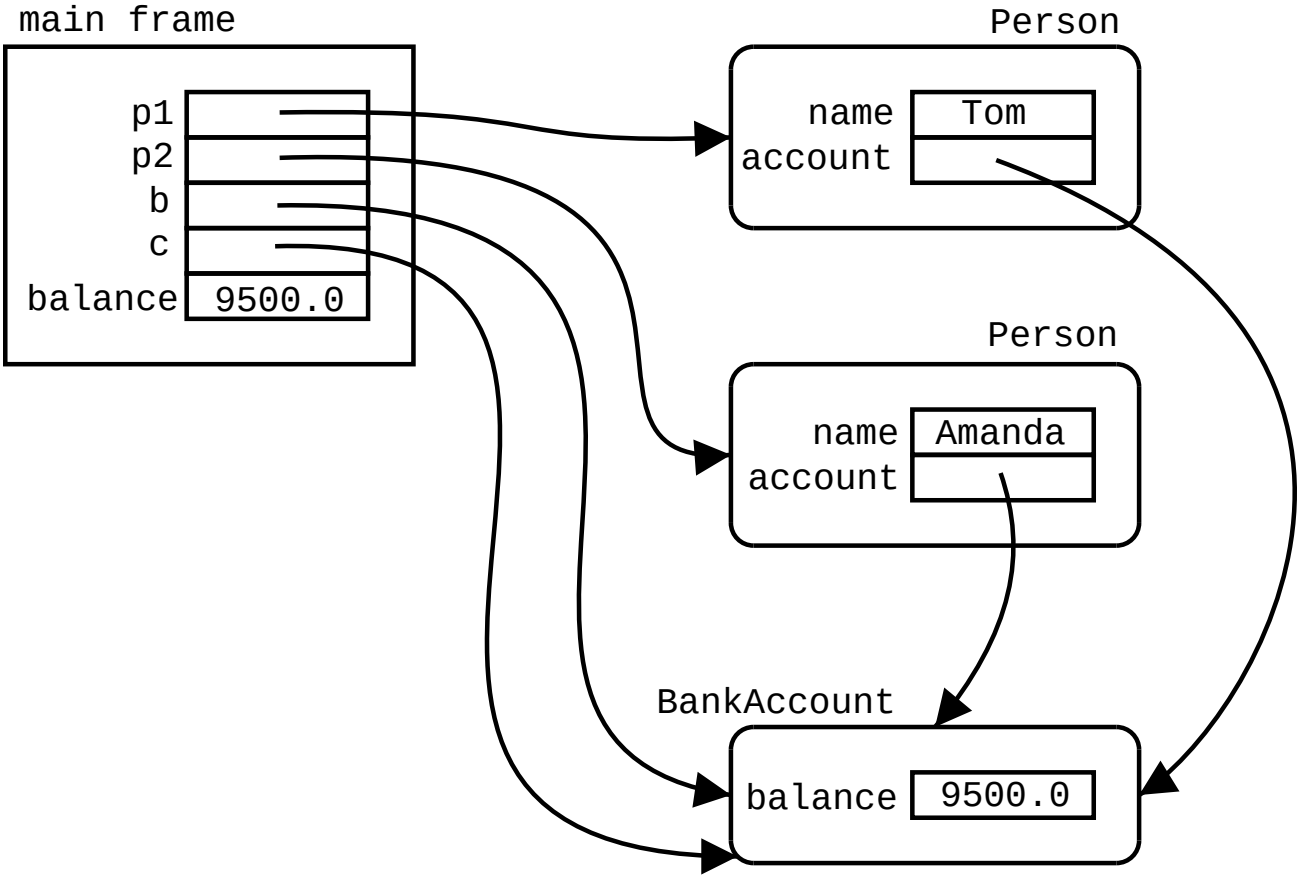
```
public class BankingTest
{
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Amanda”);
    BankAccount b = new BankAccount(10000.0f);
    p1.set_account(b);
    p2.set_account(b);

    b.withdraw(500.0f);
    BankAccount c = p2.account();
    float balance = c.balance();
    System.out.println(balance);
  }
}
```

Shared references




main frame
p1
p2
b
c
balance
Person
name
Tom
account
null
Person
name
Amanda
account
null
BankAccount
balance
10000.0


main frame
p1
p2
b
c
balance
Person
name
Tom
account
Person
name
Amanda
account
BankAccount
balance
10000.0


main frame
p1
p2
b
c
balance
9500.0
Person
name
Tom
account
Person
name
Amanda
account
BankAccount
balance
9500.0

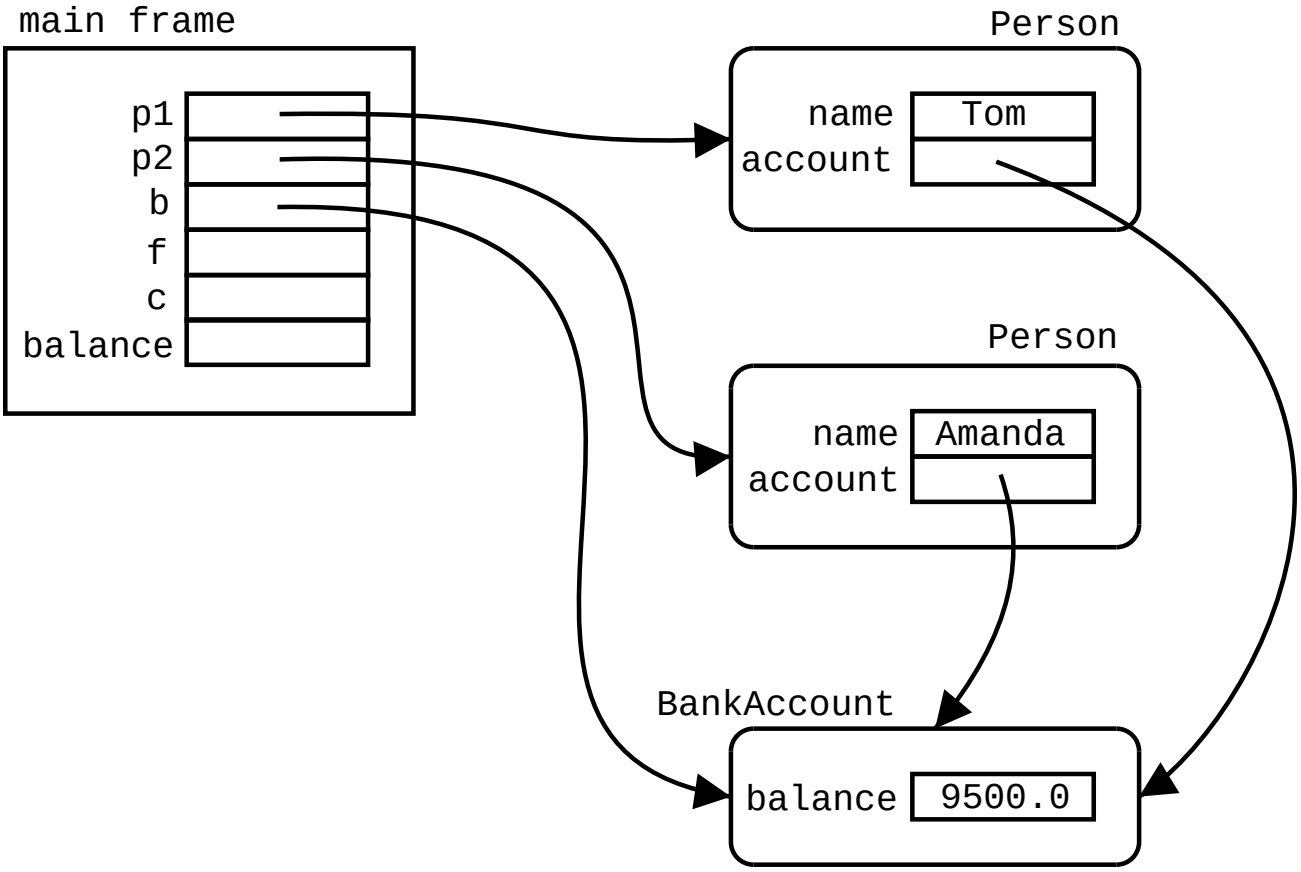
Shared references vs static variables

- In the BankingTest example b is shared between p1 and p2 only, not between all Person objects
- Static variables are like aliases, but they force all objects of the class to share the static reference, while non-static shared references are shared between specific objects.
- Furthermore, if a variable is declares as static the object it refers to is always shared between all objects in the class, while a non-static shared reference might become "unshared".

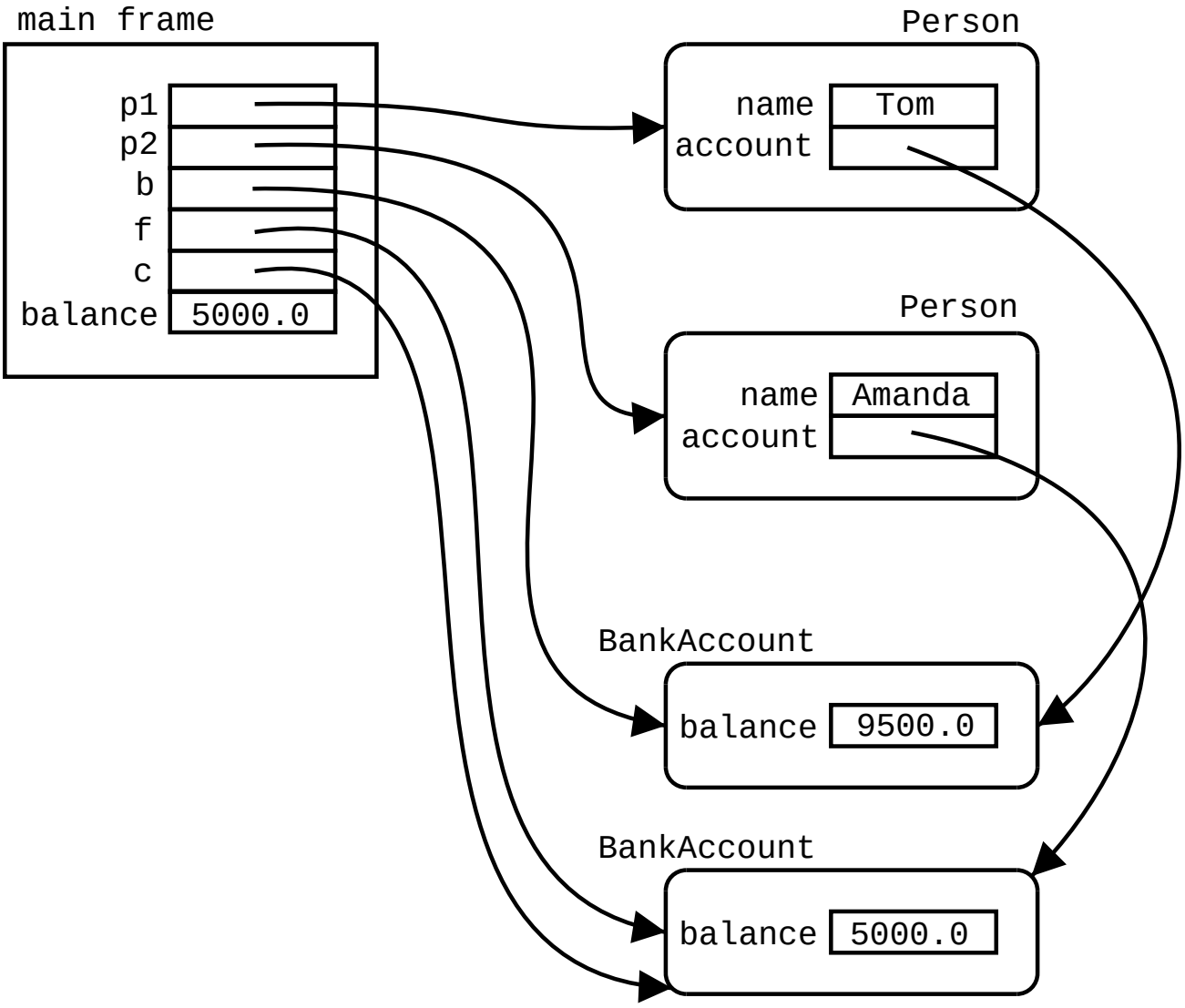
Shared references vs static variables

```
public class BankingTest
{
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Amanda”);
    BankAccount b = new BankAccount(10000.0f);
    p1.set_account(b);
    p2.set_account(b);
    b.withdraw(500.0f);
    BankAccount f = new BankAccount(5000.0f);
    p2.set_account(f);
    BankAccount c = p2.account();
    float balance = c.balance();
    System.out.println(balance);
  }
}
```

Shared references vs static variables



Shared references vs static variables



Pointer equality

- Pointer equality also called "physical" equality is equality (sameness) of references.
- The == operator is used for testing for pointer equality.
- Pointer equality is used to test for sameness of objects:

```
A x, y;
x = new A();
y = x;
```

- ...then `x == y` is true, but in

```
A x, y;
x = new A();
y = new A();
```

- ... `x == y` is false, even if the attributes of the objects are the same.
- Pointer equality is an equivalence between objects of the same class only.

Example

```
public class BankingTest
{
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Amanda”);
    BankAccount b = new BankAccount(10000.0f);
    p1.set_account(b);
    p2.set_account(b);

    BankAccount d = p1.account();
    d.withdraw(500.0f);
    BankAccount c = p2.account();
    if (c == d)
      System.out.println(“It’s a shared account”);
  }
}
```

Being equal to something

- Structural equality: when the aggregates (parts) of two different objects are equal
- Structural equality is only between objects of the same class.
- Two objects are structurally equal if their attributes are equal
- Suppose we have a class

```
class A {
    String x, y;
    A(String x, String y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Being equal to something

- and there is some client with

```
A a1 = new A(“hello”, “bye”);
A a2 = new A(“hello”, “bye”);
A a3 = new A(“bonjour”, “bye”);
```

- then `a1` is structurally equal to `a2`, but `a3` is not structurally equal to either `a1` or `a2`.

- If we want to test for structural equality we must explicitely provide the code. This is usually done by writing a method called “equal” or “equals”:

Structural equality

```
class A {
    String x, y;
    A(String x, String y)
    {
        this.x = x;
        this.y = y;
    }
    boolean equals(A other)
    {
        return this.x == other.a
            && this.y == other.y;
    }
}
```

Structural equality

```
public class Test
{
  public static void main(String[] args)
  {
    A a1 = new A("hello", "bye");
    A a2 = new A("hello", "bye");
    A a3 = new A("bonjour", "bye");
    if (a1.equals(a2))
      System.out.println(“a1 is equal to a2”);
    if (a2.equals(a3))
      System.out.println(“a2 is equal to a3”);
    if (a1 == a2)
      System.out.println(“a1 is the same as s2”);
  }
}
```

Structural equality vs pointer equality

- Note that
  - If two objects are the same (equal by pointer equality) then they are (structurally) equal, ...
    This is, x == y implies that x.`equals`(y) must evaluate to true.
  - ...but if two objects are structurally equal, they may not be physically the same.
    This is, it may be the case that x.`equals`(y) evaluates to true, but x == y may be false.

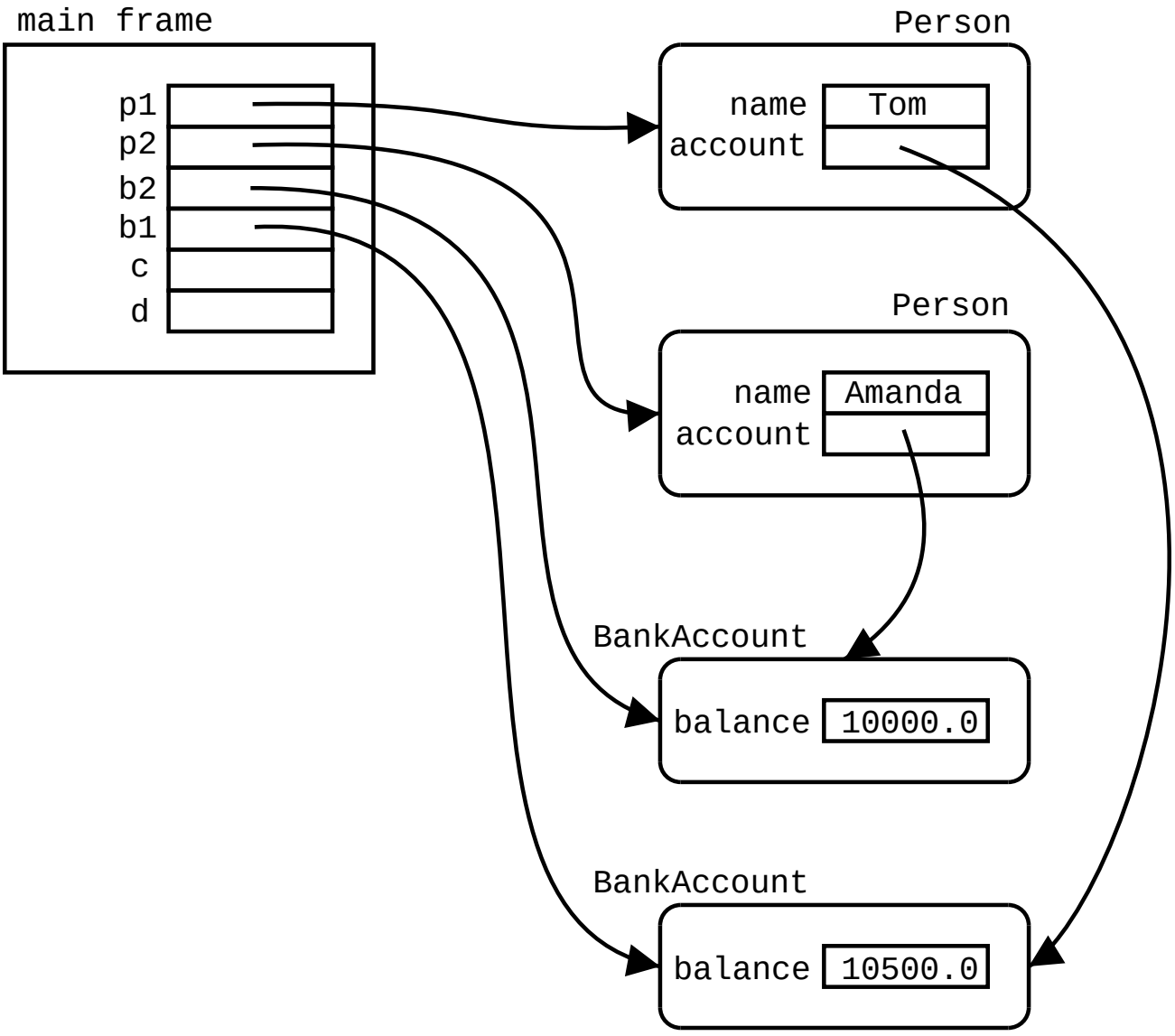
Example

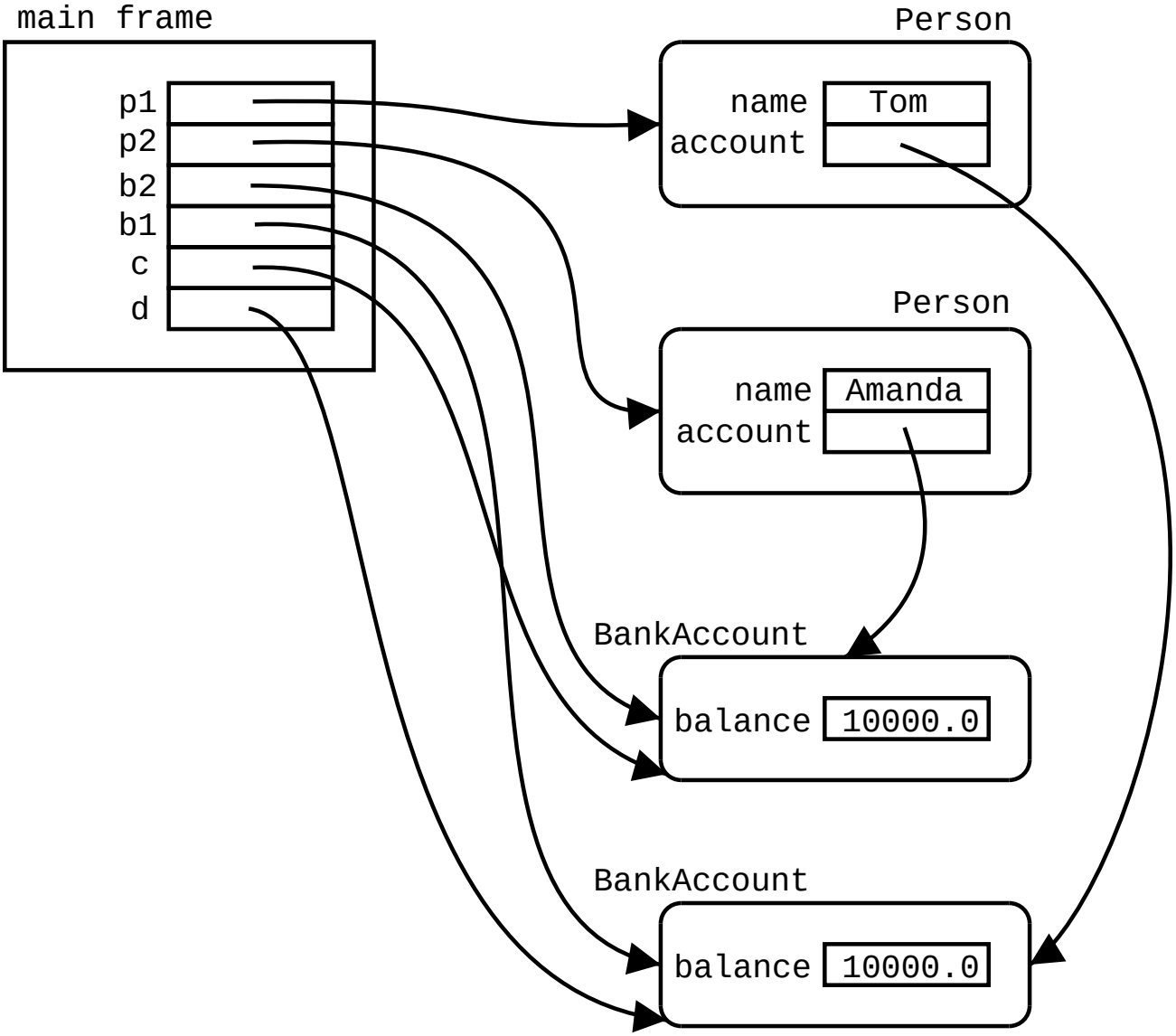
```
public class BankAccount {
  private float balance;
  // ... same as before
  public boolean equals(BankAccount other_account)
  {
    return this.balance == other_account.balance;
  }
}
```

Example

```
public class BankingTest
{
  public static void main(String[] args)
  {
    Person p1 = new Person(“Tom”);
    Person p2 = new Person(“Amanda”);
    BankAccount b1 = new BankAccount(10500.0f);
    BankAccount b2 = new BankAccount(10000.0f);
    p1.set_account(b1);
    p2.set_account(b2);
    BankAccount d = p1.account();
    d.withdraw(500.0f);
    BankAccount c = p2.account();
    if (c.equals(d))
      System.out.println(“They are equal accounts”
  }
}
```

Example


main frame
p1
p2
b2
b1
c
d
Person
name
Tom
account
Person
name
Amanda
account
BankAccount
balance
10000.0
BankAccount
balance
10500.0


main frame
p1
p2
b2
b1
c
d
Person
name
Tom
account
Person
name
Amanda
account
BankAccount
balance
10000.0
BankAccount
balance
10000.0